Design and Analysis of Algorithms (21CS42)

General method with Examples

Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

Examples:

1. **Knapsack:**

   The solution to the knapsack problem can be viewed as the result of a sequence of decisions. We have to decide the values of x_i , 1<=i<=n. first we make a decision on x_i ,1<=i<=n. first make a decision on x_1, then on x_2 , then on x_3 and so on. An optimal sequence of decisions maximizes the objective function $\sum p_i x_i$. (it also satisfies the constraints $\sum w_i x_i$<=m and 1<=x_i<=1.)

2. **Shortest path:**

   One way to find the shortest path from vertex i to vertex j in a directed graph G is to decide which vertex should be the second vertex, which is the third, and so on, until vertex j is reached. An optimal sequence of decisions is one that results in a path of least length

   For some of the problems that may be viewed in the way, an optimal sequence of decisions can be found by making the decisions one at a time and never making an erroneous decision. This is true for all problems solvable by the greedy method. For many other problems, it is not possible to make stepwise decisions in such a manner that the sequence of decisions made is optimal.

Multistage graphs

A multistage graph G=<V,E> is a directed graph in which the vertices are partitioned into k≥ 2 disjoint sets V_i, $1 \le i \le k$. in addition, if <u,v> is an edge in E, then u∈V_i and v∈V_{i+1} for some i, $1 \le i < k$. The sets V_1 and V_k. the vertex s is the source, and t the sink. Let c(I,j) be the cost of edge <I,j>. the cost of the path from s to t is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum- cost path from s to t. each set V_i defines a stage in the graph. Because of the constraints on E, every path from s to t starts in stage 1, goes to stage 2, then to stage 3, then to stage 4 and so on, and eventually terminates in stage k. fig 1 shows a five stage graph. A minimum costs to t path is indicated by the broken edges.

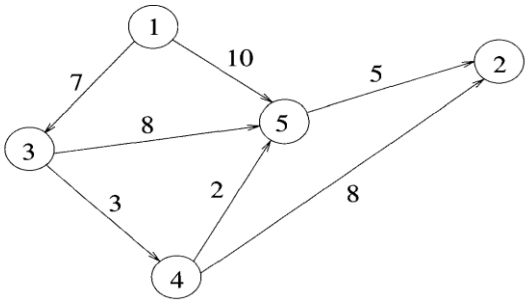


Fig 1 directed weighted graph

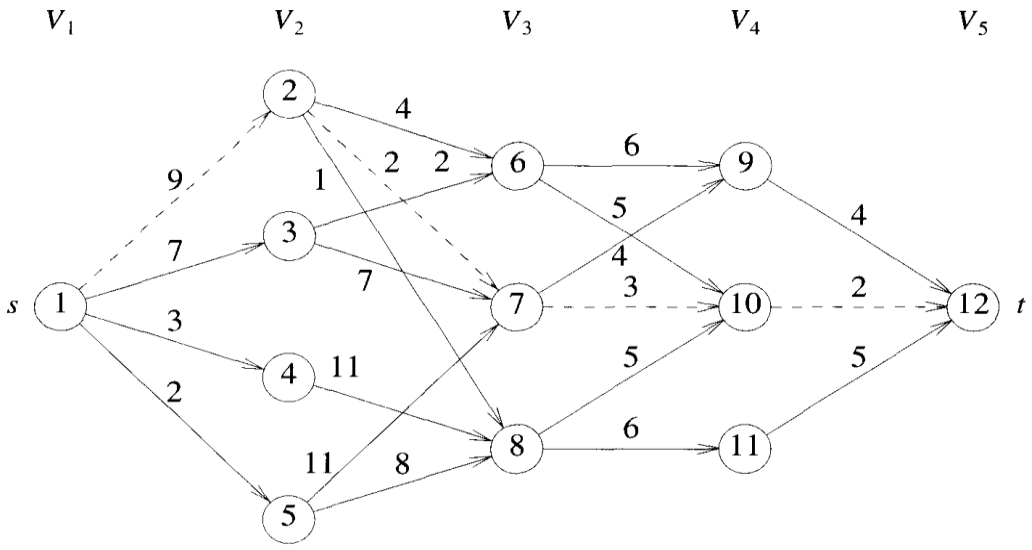


Fig 2 five stage graph

Algorithm: multistage graph pseudo code corresponding to the forward approach

Algorithm FGRAPH(G,k,n,p)

// Input: a k stage graph G=(V,E) with n vetices indexed inorder of stages. E is a set of edges and c(I,j) is the cost of(I,j) . p[1..k] is a minimum cost path.

```
{
cost[n]:=0.0;
for j:=n-1 to 1 step-1 do
{
//compute cost[j]
Let r be a vertex such that <j,r> is an edge of G and c[j,r]+cost[r] is minimum;
cost[j]:=c[j,r]+cost[r];
d[j]:=r;
}
// find minimum cost path
p[1]:=1;
p[k]:=n;
for j:=2 to k-1 do  p[j]:=d[p[j-1]];
}
```

Algorithm: multistage graph pseudo code corresponding to the backward approach

```
Algorithm BGraph(G,k,n,p)
{
//same function as FgRAPH
        bcost[1:=0.0;
        for j:=2 to n do
                {
                // compute bcost[j]
                let r such that <r,j> is an edge of bcost[j]:=bcost[r]+c[r,j];
                D[j];=r;
                }
//find a minimum cost path
p[1]:=1;p[k]:=n;
for j:=k-1 to 2 do p[j]:=d[p[j+1]];
}
```

Algorithm: multistage graph pseudo code corresponding to the backward approach

Transitive Closure

Warshall's Algorithm

Recall that the adjacency matrix $A = \{aij\}$ of a directed graph is the boolean matrix that has 1 in its *i*th row and *j*th column if and only if there is a directed edge from the *i*th vertex to the *j*th vertex. We may also be interested in a matrix containing the information about the existence of directed paths of arbitrary lengths between vertices of a given graph. Such a matrix, called the transitive closure of the digraph, would allow us to determine in constant time whether the *j*th vertex is reachable from the *i*th vertex

DEFINITION

The ***transitive closure*** of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{tij\}$, in which the element in the *i*th row and the *j*th column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the *i*th vertex to the *j*th vertex; otherwise, *tij* is 0.

An example of a digraph, its adjacency matrix, and its transitive closure is given in Figure 3. We can generate the transitive closure of a digraph with the help of depth firstsearch or breadth-first search. Performing either traversal starting at the *i*th vertex gives the information about the vertices reachable from it and hence the columns that contain 1's in the *i*th row of the transitive closure. Thus, doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.

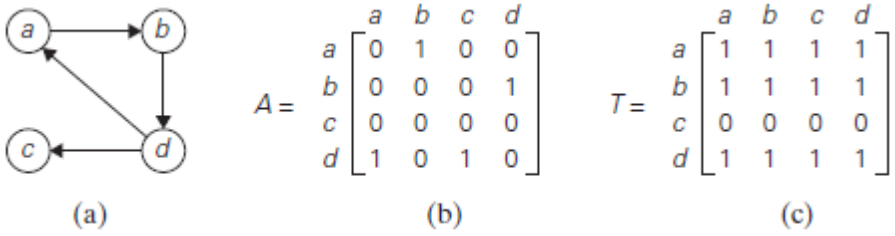


FIGURE 3 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

Since this method traverses the same digraph several times, we should hope that a better algorithm can be found. Indeed, such an algorithm exists. It is called ***Warshall's algorithm*** after Stephen Warshall, who discovered it. It is convenient to assume that the digraph's vertices and hence the rows and columns of the adjacency matrix are numbered from 1 to *n*.

Warshall's algorithm constructs the transitive closure through a series of $n \times n$ boolean matrices:

$$R^{(0)}. . . , R^{(k-1)}, R^{(k)}, . . . R^{(n)} \qquad \textit{Eq 1.}$$

Each of these matrices provides certain information about directed paths in the digraph. Specifically, the element $r^{(k)}\ ij$ in the ith row and jth column of matrix $R^{(k)}$ $(i, j = 1, 2, . . . , n, k = 0, 1, . . . , n)$ is equal to 1 if and only if there exists a directed path of a positive length from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k. Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph. (Recall that the adjacency matrix contains the information about one-edge paths, i.e., paths with no intermediate vertices.) $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate; thus, with more freedom, so to speak, it may contain more 1's than $R^{(0)}$. In general, each subsequent matrix in series has one more vertex to use as intermediate for its paths than its predecessor and hence may, but does not have to, contain more 1's. The last matrix in the series, $R^{(n)}$reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

The central point of the algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor $R^{(k-1)}$ in series Eq 1. Let $R^{(k)}\ ij$, the element in the ith row and jth column of matrix $R^{(k)}$, be equal to 1. This means that there exists a path from the ith vertex vi to the jth vertex vj with each intermediate vertex numbered not higher than k:

vi, a list of intermediate vertices each numbered not higher than k, vj. *Eq 2*

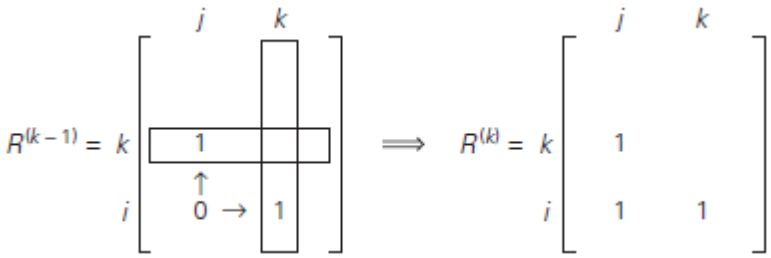


FIGURE 4 Rule for changing zeros in Warshall's algorithm.

Two situations regarding this path are possible. In the first, the list of its intermediate vertices does not contain the kth vertex. Then this path from vi to vj has intermediate vertices numbered not higher than $k - 1$, and therefore $r^{(k-1)}\ ij$ is equal to 1 as well. The second possibility is that path Eq 2 does contain the kth vertex vk among the intermediate vertices.

Without loss of generality, we may assume that *vk* occurs only once in that list. (If it is not the case, we can create a new path from *vi* to *vj* with this property by simply eliminating all the vertices between the first and last occurrences of *vk* in it.) With this caveat, path (Eq 2) can be rewritten as follows:

vi, vertices numbered ≤ *k* − 1, *vk,* vertices numbered ≤ *k* − 1, *vj* .

The first part of this representation means that there exists a path from *vi* to *vk* with each intermediate vertex numbered not higher than *k* − 1 (hence, $r^{(k-1)}$ *ik* = 1*),* and the second part means that there exists a path from *vk* to *vj* with each intermediate vertex numbered not higher than *k* − 1 (hence, $r^{(k-1)}$ *kj*= 1*).*

What we have just proved is that if $r^{(k)}ij$ = 1, then either $r^{(k-1)}ij$= 1 or both $r^{(k-1)}$ *ik*= 1 and $r^{(k-1)}kj$= 1. It is easy to see that the converse of this assertion is also true. Thus, we have the following formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$: $r^{(k)}$ *ij*= $r^{(k-1)}ij$ OR ($r^{(k-1)}{}_{ik}$ AND $r^{(k-1)}kj$) Eq 3

Formula in eq 3 is at the heart of Warshall's algorithm. This formula implies the following rule for generating elements of matrix $R^{(k)}$ from elements of matrix $R^{(k-1)}$ which is particularly convenient for applying Warshall's algorithm by hand: If an element *rij* is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$. If an element *rij* is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row *i* and column *k* and the element in its column *j* and row *k* are both 1's in $R^{(k-1)}$. This rule is illustrated in Figure 4.

As an example, the application of Warshall's algorithm to the digraph in Figure 3 is shown in Figure 5.

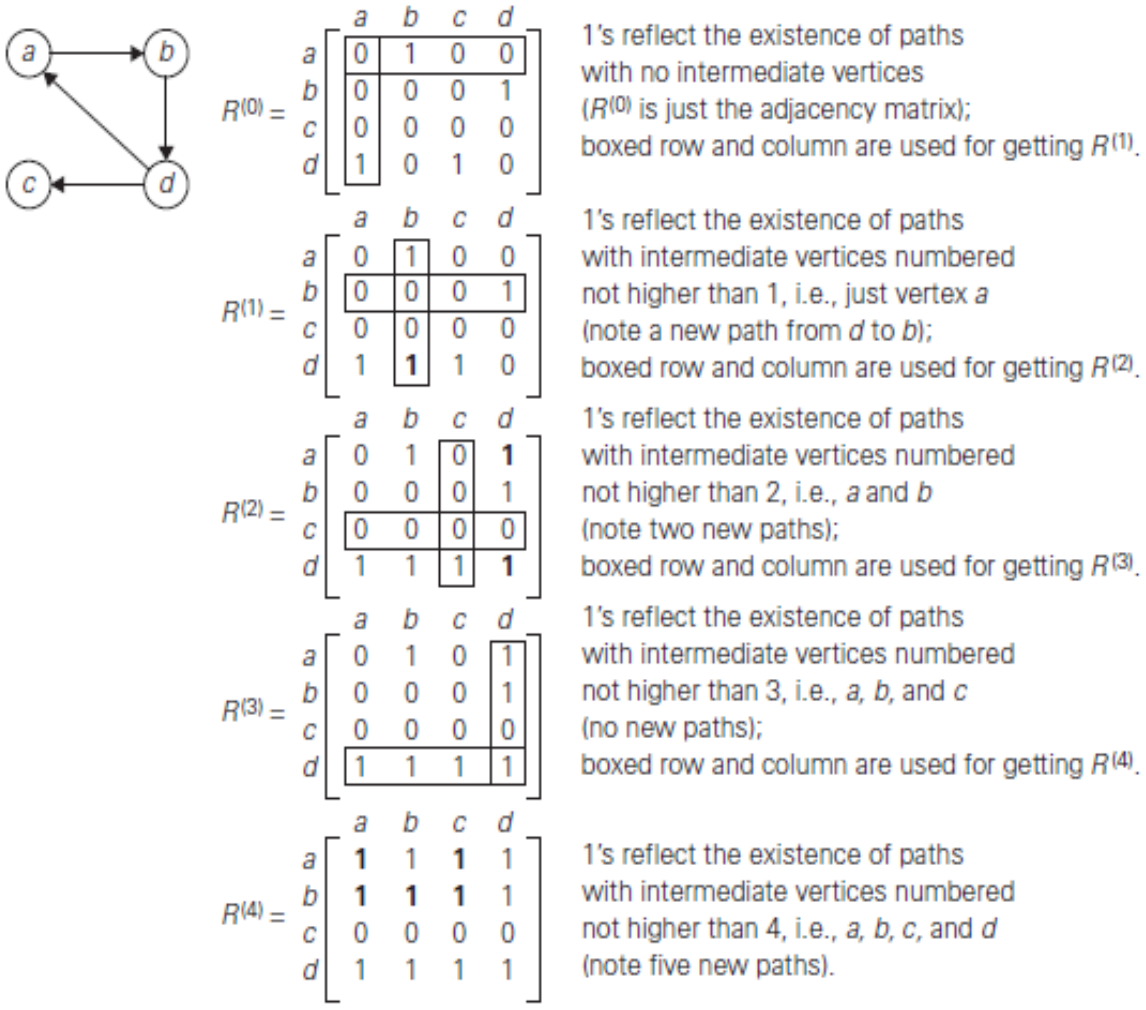


FIGURE 5 Application of Warshall's algorithm to the digraph shown. New 1's are in bold.

Here is pseudocode of Warshall's algorithm.

ALGORITHM *Warshall(A*[1..*n*, 1..*n*]*)*

//ImplementsWarshall's algorithm for computing the transitive closure

//Input: The adjacency matrix *A* of a digraph with *n* vertices

//Output: The transitive closure of the digraph

$R(0) \leftarrow A$

for $k \leftarrow 1$ **to** n **do**

for $i \leftarrow 1$ **to** n **do**

for $j \leftarrow 1$ **to** n **do**

$R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$

return $R^{(n)}$

Several observations need to be made about Warshall's algorithm. First, its time efficiency is only $\theta(n^3)$. In fact, for sparse graphs represented by their adjacency lists, the traversal-based algorithm

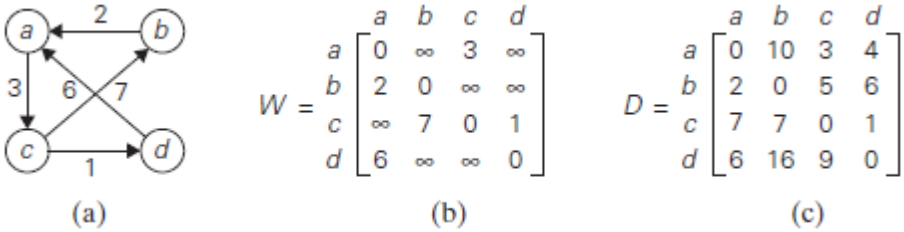


FIGURE 6(a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

All Pairs Shortest Paths

Floyd's Algorithm

Given a weighted connected graph (undirected or directed), the ***all-pairs shortest paths problem*** asks to find the distances—i.e., the lengths of the shortest paths—from each vertex to all other vertices. This is one of several variations of the problem involving shortest paths in graphs. Because of its important applications to communications, transportation networks, and operations research, it has been thoroughly studied over the years. Among recent applications of the all-pairs shortest-path problem is precomputing distances for motion planning in computer games. It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix***: the element dij in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex. For an example, see Figure 4

We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm. It is called ***Floyd's algorithm*** after its co-inventor Robert W. Floyd. It is applicable to both undirected and directed weighted graphs provided that they do not contain a cycle of a negative length. (The distance between any two vertices in such a cycle can be made arbitrarily small by repeating the cycle enough times.) The algorithm can be enhanced to find not only the lengths of the shortest paths for all vertex pairs but also the shortest paths themselves.

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:

$$D(0), \ldots, D(k-1), D(k), \ldots, D(n). \quad \textbf{(4)}$$

Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix in question. Specifically, the element $d(k)\ ij$ in the ith row and the jth column of matrix $D(k)\ (i, j = 1, 2, \ldots, n, k = 0, 1, \ldots, n)$ is equal to the length of the shortest path among all paths from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k. In particular, the series starts with $D(0)$, which does not allow any intermediate vertices in its paths; hence, $D(0)$ is simply the weight matrix of the graph. The last matrix in the series, $D(n)$, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate and hence is nothing other than the distance matrix being sought.

As in Warshall's algorithm, we can compute all the elements of each matrix $D(k)$ from its immediate predecessor $D(k-1)$ in series . Let $d(k)\ ij$ be the element in the ith row and the jth column of matrix $D(k)$. This means that $d(k)\ ij$ is equal to the length of the shortest path among all paths from the ith vertex vi to the jth vertex vj with their intermediate vertices numbered not higher than k: vi, a list of intermediate vertices each numbered not higher than k, vj . (5)

we can partition all such paths into two disjoint subsets: those that do not use the kth vertex vk as intermediate and those that do. Since the paths of the first subset have their intermediate vertices numbered not higher than $k - 1$, the shortest of them is, by definition of our matrices, of length $d(k-1)\ ij$.

What is the length of the shortest path in the second subset? If the graph does not contain a cycle of a negative length, we can limit our attention only to the paths in the second subset that use vertex vk as their intermediate vertex exactly once (because visiting vk more than once can only increase the path's length). All such paths have the following form: vi, vertices numbered $\le k - 1$, vk, vertices numbered $\le k - 1$, vj .

In other words, each of the paths is made up of a path from vi to vk with each intermediate vertex numbered not higher than $k - 1$ and a path from vk to vj with each intermediate vertex numbered not higher than $k - 1$. The situation is depicted symbolically in Figure 7.

Since the length of the shortest path from vi to vk among the paths that use intermediate vertices numbered not higher than $k - 1$ is equal to $d^{(k-1)}\ ik$ and the length of the shortest

path from *vk* to *vj* among the paths that use intermediate vertices numbered not higher than $k - 1$is equal to $d^{(k-1)}kj$, the length of the shortest path among the paths that use the *k*th vertex is equal to $d^{(k-1)}ik + d^{(k-1)}kj$. Taking into account the lengths of the shortest paths in both subsets leads to the following recurrence:

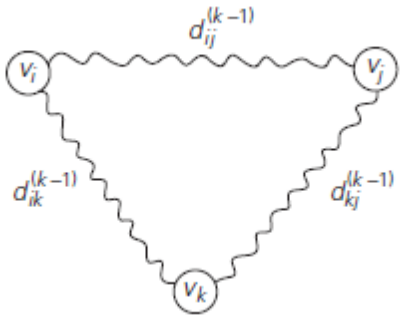


FIGURE 7 Underlying idea of Floyd's algorithm.

$$d(k)ij = \min\{d(k-1)ij\ ,\ d(k-1)\ ik + d(k-1)kj\} \text{ for } k \geq 1,\ d(0)ij = wij\ . \quad \textbf{(6)}$$

To put it another way, the element in row *i* and column *j* of the current distance matrix *D(k−1)* is replaced by the sum of the elements in the same row *i* and the column *k* and in the same column *j* and the row *k* if and only if the latter sum is smaller than its current value. The application of Floyd's algorithm to the graph in Figure 4 is illustrated in Figure 8. Here is pseudo code of Floyd's algorithm. It takes advantage of the fact that the next matrix in sequence (6) can be written over its predecessor.

ALGORITHM *Floyd(W*[1*..n,* 1*..n*]*)*

//Implements Floyd's algorithm for the all-pairs shortest-paths problem

//Input: The weight matrix *W* of a graph with no negative-length cycle

//Output: The distance matrix of the shortest paths' lengths

D ←*W* //is not necessary if *W* can be overwritten

for *k*←1 **to** *n* **do**

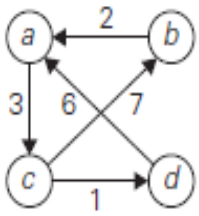
for *i* ←1 **to** *n* **do**

for *j* ←1 **to** *n* **do**

$D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$

return *D*

Obviously, the time efficiency of Floyd's algorithm is cubic—as is the time efficiency of Warshall's algorithm. In the next chapter, we examine Dijkstra's algorithm—another method for finding shortest paths.



$D^{(0)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$D^{(3)}$ =

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)}$ =

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

FIGURE 8 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

Knapsack problem

We start this section with designing a dynamic programming algorithm for the knapsack problem: given n items of known weights $w1, \ldots, wn$ and values $v1, \ldots, vn$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. (This problem was introduced in Section 3.4, where we discussed solving it by exhaustive search.) We assume here that all the weights and the knapsack capacity are positive integers; the item values do not have to be integers. To design a dynamic programming algorithm , we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms first i items, $1 \le i \le n$, with weights $w1, \ldots, wi$, values $v1, \ldots, vi$, and knapsack capacity j, $1 \le j \le W$. Let $F(i, j)$ be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the ith item and those that do. Note the following:

1. Among the subsets that do not include the ith item, the value of an optimal subset is, by definition, $F(i - 1, j)$.

2. Among the subsets that do include the ith item (hence, $j - wi \ge 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ items that fits into the knapsack of capacity $j - wi$. The value of such an optimal subset is $vi + F(i - 1, j - wi)$.

Thus, the value of an optimal solution among all feasible subsets of the first I items is the maximum of these two values. Of course, if the ith item does not fit into the knapsack, the value of an optimal subset selected from the first i items is the same as the value of an optimal subset selected from the first $i - 1$ items. These observations lead to the following recurrence:

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \ge 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

It is convenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \ge 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \ge 0.$$

that fit into the knapsack of capacity *W*, and an optimal subset itself. Figure 13 illustrates the values involved in equations (8.6) and (8.7). For *i, j* > 0, to compute the entry in the *i*th row and the *j*th column, *F(i, j)*, we compute the maximum of the entry in the previous row and the same column and the sum of *vi* and the entry in the previous row and *wi* columns to the left. The table can be filled either row by row or column by column.

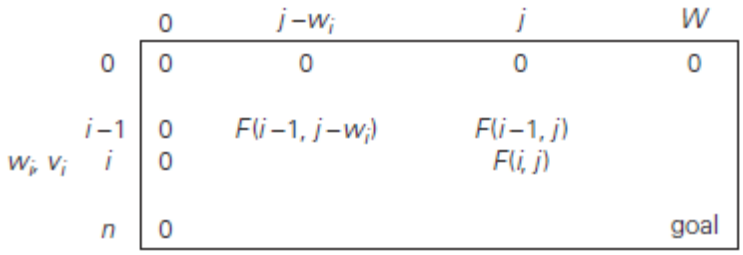


FIGURE 13 Table for solving the knapsack problem by dynamic programming

EXAMPLE 1 Let us consider the instance given by the following data:

item weight value

1 2 $12

2 1 $10 capacity *W* = 5.

3 3 $20

4 2 $15

The dynamic programming table, filled by applying formulas is shown in Figure 4

Thus, the maximal value is *F(*4, 5*)* = $37. We can find the composition of an optimal subset by backtracing the computations of this entry in the table. Since *F(*4, 5*) > F(*3, 5*)*, item 4 has to be included in an optimal solution along with an optimal subset for filling 5 − 2 = 3 remaining units of the knapsack capacity. The value of the latter is *F(*3, 3*)*. Since *F(*3, 3*) = F(*2, 3*)*, item 3 need not be in an optimal subset. Since *F(*2, 3*) > F(*1, 3*)*, item 2 is a part of an optimal selection, which leaves element *F(*1, 3 − 1*)* to specify its remaining composition. Similarly, since *F(*1, 2*) > F(*0, 2*)*, item 1 is the final part of the optimal solution {item 1, item 2, item 4}.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

Figure: 14 Example of solving an instance of the knapsack problem by the dynamic programming algorithm.

Memory Functions

The following algorithm implements this idea for the knapsack problem. After initializing the table, the recursive function needs to be called with $i = n$ (the number of items) and $j = W$ (the knapsack capacity).

ALGORITHM *MFKnapsack(i, j)*

//Implements the memory function method for the knapsack problem

//Input: A nonnegative integer i indicating the number of the first items being considered and a nonnegative integer j indicating the knapsack capacity

//Output: The value of an optimal feasible subset of the first i items

//Note: Uses as global variables input arrays *Weights*[1..n], *V alues*[1..n], and table F[0..n, 0..W] whose entries are initialized with −1's except for row 0 and column 0 initialized with 0's

if $F[i, j] < 0$

if $j <$ *Weights*[i]

value←*MFKnapsack*($i - 1, j$)

else

value←max(*MFKnapsack*($i - 1, j$),

Values[i]+ *MFKnapsack*($i - 1, j -$ *Weights*[i]))

$F[i, j] \leftarrow$ *value*

return $F[i, j]$

EXAMPLE 2 Let us apply the memory function method to the instance considered in Example 1. The table in Figure 15 gives the results. Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

FIGURE 15 Example of solving an instance of the knapsack problem by the memory function algorithm.

Just one nontrivial entry, *V (*1, 2*),* is retrieved rather than being recomputed. For larger instances, the proportion of such entries can be significantly larger.

Bellman-Ford Algorithm

Single source shortest path when some or all of the edges of the directed graph G may have negative weights.

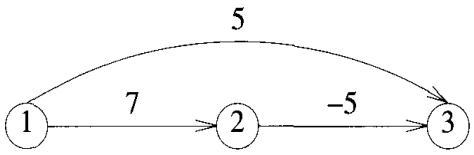


Fig 16: Directed graph with negative edge length.

When negative edge lengths are permitted, It is requires that the graph have no cycles of negative length. This is necessary to ensure that shortest paths consist of a finite number of edges. For example consider Fig 17

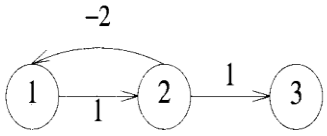


Fig 17: Graph with negative edge cycle

The length of the shortest path from vertex 1 to 3 is -∞. the length of the path 1,2,1,2,1,2,….1,2,3 can be made arbitrarily small.

let dist ^{l}[u] be the length of a shortest path from the source vertex to v to vertex u under the constraint that the shortest path contains at most l edges. then, dist^{1}[u]=cost[u,v], 1<=u<= n. if there are no negative cycles the search for shortest path can be limited to n-1 edges. Hence dist^{n-1}[u] is the length of an unrestricted shortest path from v to u.

The goal is to compute dist $^{n-1}$[u] for all u. this can be done using dynamic programming method. With the following observations:

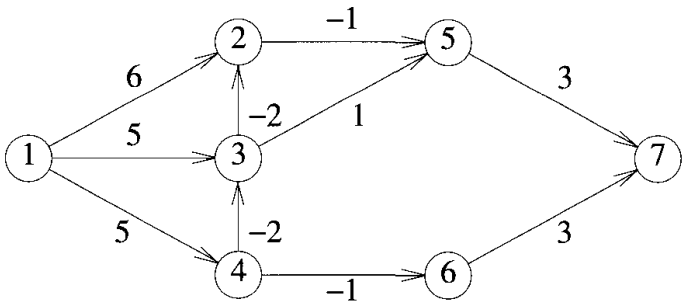
1. If the shortest path from v to u with at most k, k >1 , edges has no more than k-1 edges, then dist ^{k}[u]=dist^{k-1}[u].
2. If the shortest path from v to u with at most k, k >1, edges has exactly k edges then, it is made up of a shortest path from v to some vertex j followed by the edge<j,u>. The path from v to j has k-1 edges, and its length is dist^{k-1}[j]. all vertices i such that the edge<i, u> is in the graph are candidates for j, since shortest path is to be calculated, the I that minimizes dist $^{k-1}$ [i] + cost[i,u] is the correct value for j.

These observations result in the following recurrence for dist:

dist ^{k}[u]=min{ dist $^{k-1}$[u], min $_{i}$ {dist $^{k-1}$[i]+cost[i,u]}

This recurrence can be used to compute dist ^{k} from dist $^{k-1}$,for k=2,3,…. n-1.

Example 1:



(a) A directed graph

	$dist^k[1..7]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Fig 18: Shortest paths with negative edge lengths

```
Algorithm bellman ford(v,cost,dist,n)
//single source/ all destination shortest paths with negative edhe costs
{
for i:=1 to n do
                dist[i]:=cost[v,i];
for k:=2 to n-1 do
    for each u such that u≠v and u has at least one incoming edge do
        for each <i,u> in the graph do
            if dist[u]>dist[i]+cost[I,u] then
                dist[u]:=dist[i]+cost[i,u];
}
```

Algorithm for bellman ford to compute shortest path with general weights.

8 Travelling Sales Person problem

Let G= (V,E) be a directed graph with edge costs c_{ij}. The variable cij is defined such that $c_{ij}>0$ for all I and j and $c_{ij}=\infty$ if <i,j>∉ E. let |V| =n and assume n>1. A tour of G is a directed

simple cycle that includes every vertex in V. the cost of a tour is the sum of the cost of the edges on the tour. The travelling salesperson problem is to find a tour of minimum cost.

The travelling salesperson problem finds application in a variety of situations. Suppose we have to route a postal van to pick up mail from mail boxes located in n different sites. An n+1 vertex graph can be used to represent the situation. one vertex represents the post office from which the postal van starts and to which it must return. Edge <i,j> is assigned a cost equal to the distance from site I to site j . The route taken by the postal van is tour, and to calculate the minimum tour.

let g(I,S) be the length of a shortest path atarting at vertex I, going through all vertices in the S, and terminating at vertex 1. the function g(1,V-{1}) I the length of an optimal salesperson tour. from the principal of optimality it follows that an optimal salesperson tour. from the principal of optimality it follows that

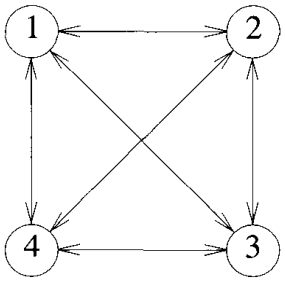
g(1,V-{1})= $\min_{2\le K\le n}\{c_{1k} + g(k, V-\{1,K\})\}$ eq1

By generalizing we obtain (for i∉S)

g(i, S)=$\min_{j\in S}\{c_{ij} + g(j, S-\{j\})\}$ eq2

Example 1

Consider the directed graph of fig 19(a). The edge length are given in the matrix c of fig 19 (b)



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

fig 19: Directed graph with edge length matrix c

Thus,

g(2,∅)=c21=5 , g(3, ∅)=c31=6 and g(4, ∅)=c41=8

using eq 2 we obtain

g(2,{3})=c23+g(3, ∅)=15

g(2,{4})=18

g(3,{2})=18

g(3,{4})=20

g(4,{2})=13

g(4,{3})=15

Next we compute g(I,S) with |S|=2,i≠S and i∉S

g(2,{3,4})=min(c23+g(3,{4}),c24+g(4,{3})}=25

g(3,{2,4})=min(c32+g(2,{4}),c34+g(4,{2})}=25

g(4,{2,3})=min(c42+g(2,{3}),c43+g(3,{2})}=23

Finally from eq 1 we obtain

g(1,{2,3,4})=min9c12+g(2,{3,4}),c13+g(3,{2,4}),c14+g(4,{2,3})}

=min{35,40,43}

=35

The optimal tour of the graph has length 35. A tour of this length can be constructed if we retain with each g(I,S) be this value. Then J(1,{2,3,4})=2. Thus the tour starts from 1 and goes to 2. the remaining tour can be obtained from g(2,{3,4{). so J(2,{3,4})=4. thus the next edge is<2,4>. the remaining tour is for g(4,{3}). do J(4,{3})=3. the optimal tour is 1 ,2 ,4 ,3 ,1.

Space and Time Trade-Offs

Sorting by Counting

As a first example of applying the input-enhancement technique, we discuss its application to the sorting problem. One rather obvious idea is to count, for each element of a list to be sorted, the total number of elements smaller than this element and record the results in a table. These numbers will indicate the positions of the elements in the sorted list: e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array. Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list. This algorithm is called ***comparison counting sort***

Array $A[0..5]$		62	31	84	96	19	47
Initially	$Count[]$	0	0	0	0	0	0
After pass $i = 0$	$Count[]$	3	0	1	1	0	0
After pass $i = 1$	$Count[]$		1	2	2	0	1
After pass $i = 2$	$Count[]$			4	3	0	1
After pass $i = 3$	$Count[]$				5	0	1
After pass $i = 4$	$Count[]$					0	2
Final state	$Count[]$	3	1	4	5	0	2
Array $S[0..5]$		19	31	47	62	84	96

ALGORITHM *ComparisonCountingSort(A[0..n − 1])*
//Sorts an array by comparison counting
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $S[0..n-1]$ of A's elements sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-1$ **do** $Count[i] \leftarrow 0$
    for $i \leftarrow 0$ **to** $n-2$ **do**
        for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] < A[j]$
            $Count[j] \leftarrow Count[j] + 1$
        else $Count[i] \leftarrow Count[i] + 1$
for $i \leftarrow 0$ **to** $n-1$ **do**
    $S[Count[i]] \leftarrow A[i]$
return S

What is the time efficiency of this algorithm? It should be quadratic because the algorithm considers all the different pairs of an n-element array. More formally, the number of times its basic operation, the comparison $A[i]<A[j]$, is executed is equal to the sum we have encountered several times already:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}.$$

Input Enhancement in String Matching

In this section, we see how the technique of input enhancement can be applied to the problem of string matching. Recall that the problem of string matching requires finding an occurrence of a given string of m characters called the ***pattern*** in a longer string of n characters called the ***text***. We have discussed the brute-force algorithm: it simply matches corresponding pairs of characters in the pattern and the text left to right and, if a mismatch occurs, shifts the pattern one position to the right for the next trial. Since the maximum number of such trials is $n - m + 1$ and, in the worst case, m comparisons need to be made on each of them, the worst-case efficiency of the brute-force algorithm is in the $O(nm)$ class. On average, however, we should expect just a few comparisons before a pattern's shift, and for random natural-language texts, the average-case efficiency indeed turns out to be in $O(n + m)$.

Several faster algorithms have been discovered. Most of them exploit the input-enhancement idea: preprocess the pattern to get some information about it, store this information in a table, and then use this information during an actual search for the pattern in a given text. This is exactly the idea behind the two best known algorithms of this type: the Knuth-Morris-Pratt algorithm [Knu77] and the Boyer-Moore algorithm [Boy77].

The principal difference between these two algorithms lies in the way they compare characters of a pattern with their counterparts in a text: the Knuth- Morris-Pratt algorithm does it left to right, whereas the Boyer-Moore algorithm does it right to left.

Horspool's Algorithm

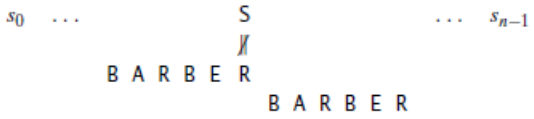
Consider, as an example, searching for the pattern BARBER in some text:

$$s0 \ldots c \ldots sn-1$$

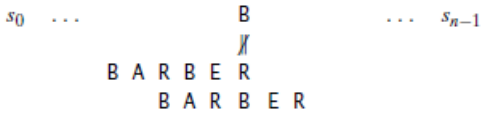
B A R B E R

Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text. If all the pattern's characters match successfully, a matching substring is found. Then the search can be either stopped altogether or continued if another occurrence of the same pattern is desired. If a mismatch occurs, we need to shift the pattern to the right. Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text. Horspool's algorithm determines the size of such a shift by looking at the character *c* of the text that is aligned against the last character of the pattern. This is the case even if character *c* itself matches its counterpart in the pattern. In general, the following four possibilities can occur.

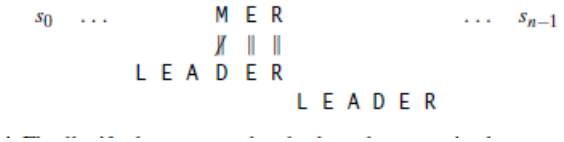
Case 1 If there are no *c*'s in the pattern—e.g., *c* is letter S in our example— we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character *c* that is known not to be in the pattern):



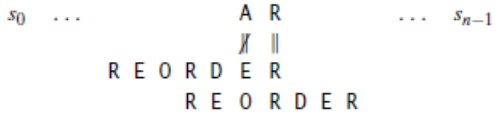
Case 2 If there are occurrences of character *c* in the pattern but it is not the last one there—e.g., *c* is letter B in our example—the shift should align the rightmost occurrence of *c* in the pattern with the *c* in the text:



Case 3 If *c* happens to be the last character in the pattern but there are no *c*'s among its other *m* − 1 characters—e.g., *c* is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length *m*:



Case 4 Finally, if *c* happens to be the last character in the pattern and there are other *c*'s among its first *m* − 1 characters—e.g., *c* is letter R in our example— the situation is similar to that of Case 2 and the rightmost occurrence of *c* among the first *m* − 1 characters in the pattern should be aligned with the text's *c*:



These examples clearly demonstrate that right-to-left character comparisons can lead to farther shifts of the pattern than the shifts by only one position always made by the brute-force algorithm. However, if such an algorithm had to check all the characters of the pattern on every trial, it would lose much of this superiority. Fortunately, the idea of input enhancement makes repetitive comparisons unnecessary. We can precompute shift sizes and store them in a table. The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters. (Note that no other information about the text in which eventual searching will be done is required.) The table's entries will indicate the shift sizes computed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

For example, for the pattern BARBER, all the table's entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively. Here is a simple algorithm for computing the shift table entries. Initialize allthe entries to the pattern's length *m* and scan the pattern left to right repeating the following step *m* − 1 times: for the *j*th character of the pattern *(*$0 \le j \le m - 2$*)*, overwrite its entry in the table with $m - 1 - j$, which is the

character's distance to the last character of the pattern. Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for the character's rightmost occurrence—exactly as we would like it to be.

ALGORITHM *ShiftTable(P* [0*..m* − 1]*)*

//Fills the shift table used by Horspool's and Boyer-Moore algorithms

//Input: Pattern *P*[0*..m* − 1] and an alphabet of possible characters

//Output: *Table*[0*..size* − 1] indexed by the alphabet's characters and

// filled with shift sizes computed by formula (7.1)

for *i* ←0 **to** *size* − 1 **do**

    Table[*i*]←*m*

for *j* ←0 **to** *m* − 2 **do**

    Table[*P*[*j*]]←*m* − 1− *j*

return *Table*

Now, we can summarize the algorithm as follows:

Horspool's algorithm

Step 1 For a given pattern of length *m* and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all *m* characters are matched (then the entry *t (c)* from the *c*'s column of the shift table where *c* is the text's character currently aligned against the last character of the pattern, and shift the pattern by *t (c)* characters to the right along the text. Here is pseudocode of Horspool's algorithm.

ALGORITHM *HorspoolMatching(P* [0..*m* − 1], *T* [0..*n* − 1]*)*

//Implements Horspool's algorithm for string matching

//Input: Pattern *P*[0..*m* − 1] and text *T* [0..*n* − 1]

//Output: The index of the left end of the first matching substring

// or −1 if there are no matches

ShiftTable(P [0..*m* − 1]*)* //generate *Table* of shifts

i ←*m* − 1 //position of the pattern's right end

while *i* ≤ *n* − 1 **do**

k←0 //number of matched characters

while *k* ≤ *m* − 1 **and** *P*[*m* − 1− *k*]= *T* [*i* − *k*] **do**

k←*k* + 1

if *k* = *m*

return *i* − *m* + 1

else *i* ←*i* + *Table*[*T* [*i*]]

return −1

EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

character *c*	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift *t*(*c*)	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R             B A R B E R
        B A R B E R         B A R B E R
          B A R B E R             B A R B E R
```

A simple example can demonstrate that the worst-case efficiency of Horspool's algorithm is in *O(nm)* But for random texts, it is in *O (n)*, and, although in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm.